Using Neo4j with Java

Learn how to interact with Neo4j using the Neo4j Java Driver

Using the driver

Introduction

In the **Cypher Fundamentals** course, you learned how to query Neo4j using Cypher.

To run Cypher statements in a Java application, you'll need the **Neo4j Java Driver**. The driver acts as a bridge between your Java code and Neo4j, handling connections to the database and the execution of Cypher queries.

Creating a Driver Instance

Open the `src/main/java/com/neo4j/app/App.java` file.

Import the driver:

Java:

```
import org.neo4j.driver.GraphDatabase;

import org.neo4j.driver.AuthTokens;
```

Creating a Driver Instance

Create a `driver` instance in `main()`:

Java:

```
public class App {

    public static void main(String[] args) {

        AppUtils.loadProperties();


        // Create a new Neo4j driver instance

        var driver = GraphDatabase.driver(

                System.getProperty("NEO4J_URI"), // (1)

                AuthTokens.basic(

                    System.getProperty("NEO4J_USERNAME"), // (2)

                    System.getProperty("NEO4J_PASSWORD"))

            );

    }

}
```

1. The connection string for your Neo4j database

2. Your Neo4j username and password

> **Best Practice**
>
> Create **one** Driver instance and share it across your entire application.

Verifying Connectivity

You can verify the connection by calling the `verifyConnectivity()` method.

Java:

```
driver.verifyConnectivity();
```

Verify Connectivity

The `verifyConnectivity()` method will raise an exception if the connection cannot be made.

Running Your First Query

The `executableQuery()` method executes a Cypher query and returns the results.

Java:

```
// (1)
var result = driver.executableQuery(
    "RETURN COUNT {()} AS count"
    ).execute();

// Get the first record
var records = result.records(); // (2)
var first = records.get(0);

// Print the count entry
System.out.println(first.get("count"));   // (3)
```

1. `executableQuery()` runs a Cypher query to get the count of all nodes in the database
2. `records()` returns a list of the records returned
3. Keys from the `RETURN` clause are accessed using the `get` method

Full driver lifecycle

Once you have finished with the driver, call `close()` to release any resources held by the driver.

Java:

```
driver.close();
```

Run the application

You can run the application to see the output:

bash:

```
./mvnw compile exec:java -Dexec.mainClass="com.neo4j.app.App"
```

> You can also run the application using the *play* button in your IDE.

Try with resources

You can use `try-with-resources` to automatically close the driver when the block is exited.

Java:

```
try (
    var driver = GraphDatabase.driver(
        System.getProperty("NEO4J_URI"),
        AuthTokens.basic(
            System.getProperty("NEO4J_USERNAME"),
            System.getProperty("NEO4J_PASSWORD"))
    )
) {
    driver.verifyConnectivity();

    var result = driver.executableQuery(
        "RETURN COUNT {()} AS count"
        ).execute();
}
```

Executing Cypher statements

Introduction

You can use the `executableQuery()` method to create one-off Cypher statements that return a small number of records.

The `execute` method fetches a list of records and loads them into memory.

Java:

```
final String cypher = """
    MATCH (p:Person {name: $name})-[r:ACTED_IN]->(m:Movie)
    RETURN m.title AS title, r.role AS role
    """;
final String name = "Tom Hanks";


var result = driver.executableQuery(cypher)
    .withParameters(Map.of("name", name))
    .execute();
```

1. The `executableQuery` method expects a Cypher statement as a string as the first argument.
2. Parameters can be passed as a map using the `withParameters()` method.
3. The `execute()` method runs the query and returns the result.

Handling the Result

The `execute()` method returns an `EagerResult` object that contains:

1. A list of `Record` objects
2. `ResultSummary` of the query execution
3. A list of keys specified in the `RETURN` clause

Java:

```
var records = result.records(); // (1)
var summary = result.summary(); // (2)
var keys = result.keys();       // (3)


System.out.println(records);
System.out.println(summary);
System.out.println(keys);
```

Accessing results

Each row returned by the query is a `Record` object. The `Record` object provides access to the data returned by the query.

You can access any item in the `RETURN` clause using the `get` method.

Java:

```
// RETURN m.title AS title, r.role AS role
var records = result.records();
records.forEach(r -> {
    System.out.println(r.get("title"));
    System.out.println(r.get("role"));
});
```

Reading and writing

By default, `executableQuery()` runs in **WRITE** mode. In a clustered environment, this sends all queries to the cluster leader, putting unnecessary load on the leader.

When you're only reading data, you can optimize performance by configuring the query to `READ` mode. This distributes your read queries across all cluster members.

Java:

```
import org.neo4j.driver.RoutingControl;


var result = driver.executableQuery(cypher)
    .withParameters(Map.of("name", name))
    .withConfig(QueryConfig.builder()
                .withRouting(RoutingControl.READ)
                .build())
    .execute();
```

Mapping Results to Java Objects

Introduction

The driver includes an **object mapping feature** that allows you to map query results directly to Java objects. This feature simplifies the process of working with Neo4j data in Java applications by eliminating the need for manual mapping with raw data types.

This feature was introduced in the driver version 5.28.5.

Object Graph Mapping (OGM)

The driver's new object mapping is not a full-fledged OGM solution. For a more comprehensive option, check out the **Neo4j OGM library**.

Domain Model

Nodes are represented by domain classes.

For the `Person` domain class, create a new file in `src/main/java/com/neo4j/app` and name it `Person.java`.

> This example will use Java records (rather than class), but classes work similarly.

Java:

```
public record Person(String id,
                     String name) {
}
```

Querying and Mapping Results

Open the `src/main/java/com/neo4j/app/App.java` file and add a method to query for a person by name and return an entity mapped to your `Person` record.

Querying and Mapping Results

Java:

```
final String personCypher = """
    MATCH (person:Person {name: $name})
    RETURN person
    """;
final String name = "Tom Hanks";


var person = driver.executableQuery(personCypher)
    .withParameters(Map.of("name", name))
    .execute()
    .records()
    .stream()
    .map(record -> record.get("person").as(Person.class)) // (1)
    .findFirst()
    .orElseThrow(() -> new RuntimeException("Person not found")); // (2)


System.out.println(person); // (3)
```

1. `.as(Person.class)` returns a single record mapped to the `Person` class
2. `.findFirst().orElseThrow()` returns the first record or throws an exception if not found
3. Print the person object to the console

Adding a Connecting Node

You can return a graph by adding the `Movie` node to the domain and connecting it to the `Person` node.

Java:

```
public record Movie(String id,
                    String title,
                    List<Person> actors) {
}
```

Querying and Returning a Graph

Queries **must** return the results that *match the domain model's structure*.

This query returns a single `Movie` node with a list of `Person` nodes as actors. The query uses the `COLLECT` clause to gather the actors into a list.

Java:

```
final String movieCypher = """
    MATCH (movie:Movie)
    LIMIT 1
    RETURN movie {
        .*,
        actors: COLLECT {
            MATCH (actor:Person)-[r:ACTED_IN]->(movie)
            RETURN actor
        }
    }
    """;
```

Mapping Results

Results are mapped to the `Movie` class using the `.as(Movie.class)` method - automatically mapping the `actors` list to a list of `Person` objects.

Missing properties

Only the `movieId` and `title` are defined in the domain class so only those are mapped from the node in Neo4j. If you want to include more properties, you can add them to the class definition.

Java:

```
var movie = driver.executableQuery(movieCypher)
            .execute()
            .records()
            .stream()
            .map(record -> record.get("movie").as(Movie.class))
            .findFirst()
            .orElseThrow(() -> new RuntimeException("No movie found"));


System.out.println(movie);
```

Graph types

Introduction

Let’s take a look at the types of data returned by a Cypher query.

The majority of the types returned by a Cypher query are mapped directly to Java types, but some more complex types need special handling.

- Graph types - Nodes, Relationships and Paths
- Spatial types - Points and distances

> **Types in Neo4j Browser**
>
> When graph types are returned by a query, they are visualized in a graph layout.

Table 1. Direct mapping

Java Type	Neo4j Cypher Type
`null`	`null`
`Boolean`	`Boolean`
`Long`	`Integer`
`Double`	`Float`
`String`	`String`
`List`	`List`
`Map`	`Map`

Graph types

The following code snippet finds all movies with the specified title and returns `person`, `acted_in` and `movie`.

Java: **Return Nodes and Relationships**

```
final String cypher = """
    MATCH path = (person:Person)-[actedIn:ACTED_IN]->(movie:Movie {title: $title})
    RETURN path, person, actedIn, movie
    """;
final String title = "Toy Story";


var result = driver.executableQuery(cypher)
    .withParameters(Map.of("title", title))
    .execute();
```

Nodes

Nodes are returned as a `Node` object.

Java: Working with Node Objects

```
import org.neo4j.driver.types.Node;


var records = result.records();
records.forEach(r -> {
    Node node = r.get("person").asNode();
});
```

Nodes

Java:

```
records.forEach(r -> {
    Node node = r.get("person").asNode();

    System.out.println(node.elementId()); // (1)
    System.out.println(node.labels());    // (2)
    System.out.println(node.values());    // (3)

    System.out.println(node.get("name")); // (4)
});
```

1. The `elementId()` method provides access to the node's element ID
   eg. `4:97b72e9c-ae4d-427c-96ff-8858ecf16f88:0`
2. The `labels()` method contains a list of labels attributed to the Node
   eg. `['Person', 'Actor']`
3. The `values()` method provides access to the node's properties as an iterable of `Value` objects.
   eg. `{name: 'Tom Hanks', tmdbId: '31'}`
4. A single property can be retrieved using the `get()` method.

Relationships

Relationships are returned as a `Relationship` object.

Java:

```
import org.neo4j.driver.types.Relationship;


records.forEach(r -> {
    Relationship actedIn = r.get("actedIn").asRelationship();


    System.out.println(actedIn.elementId()); // (1)
    System.out.println(actedIn.type()); // (2)
    System.out.println(actedIn.values()); // (3)


    System.out.println(actedIn.get("role")); // (4)


    System.out.println(actedIn.startNodeElementId()); // (5)
    System.out.println(actedIn.endNodeElementId()); // (6)
});
```

1. `elementId()` - The element ID of the relationship
   eg. `5:1218f598-63ab-460f-ac59-36d4cadee840:167495`
2. `type()` - Type of relationship
   eg. `ACTED_IN`
3. `values()` - Returns relationship properties as name-value pairs (eg. `{role: 'Woody'}`)
4. Access properties using the `get()` method
5. `startNodeElementId` - The element ID of the `Node` at the start of the relationship
6. `endNodeElementId` - The element ID of the `Node` at the end of the relationship

Paths

A path is a sequence of nodes and relationships and is returned as a `Path` object.

Java:

```
import org.neo4j.driver.types.Path;


records.forEach(r -> {
    Path path = r.get("path").asPath();


    System.out.println(path.nodes()); // (1)
    System.out.println(path.relationships()); // (2)


    System.out.println(path.start()); // (3)
    System.out.println(path.end()); // (4)
    System.out.println(path.length()); // (5)
});
```

1. `nodes()` - An iterable of `Node` objects in the path
2. `relationships()` - An iterable of `Relationship` objects in the path
3. `start()` - The `Node` object at the start of the path
4. `end()` - The `Node` object at the end of the path
5. `length()` - The number of relationships within the path

Dates and times

Temporal types

Temporal types in Neo4j are a combination of date, time and timezone elements.

Table 1. Temporal Types

Type	Description	Date?	Time?	Timezone?
`Date`	A tuple of Year, Month and Day	Y		
`Time`	The time of the day with a UTC offset	Y	Y	
`LocalTime`	A time without a timezone		Y	
`DateTime`	A combination of Date and Time	Y	Y	Y
`LocalDateTime`	A combination of Date and Time without a timezone	Y	Y	

Writing temporal types

java:

```
import java.time.ZonedDateTime;
import java.time.ZoneId;
String dtstring="2024-05-15T14:30:00+02:00";
var datetime = ZonedDateTime.of(2024, 05, 15, 14, 30, 00, 0, ZoneId.of("+02:00"));
var result = driver.executableQuery("""
    CREATE (e:Event {
        startsAt: $datetime,              // (1)
        createdAt: datetime($dtstring),   // (2)
        updatedAt: datetime()             // (3)
    })
    RETURN e.startsAt AS startsAt, e.createdAt AS createdAt, e.updatedAt AS updatedAt;
    """)
    .withParameters(
        Map.of( "datetime", datetime, "dtstring", dtstring )) // (4)
    .execute();
```

When you write temporal types to the database, you can pass the object as a parameter to the query or cast the value within a Cypher statement.

This example demonstrates how to:

1. Use a `DateTime` object as a parameter to the query (`<4>`)
2. Cast an **ISO 8601 format string** within a Cypher statement
3. Get the current date and time using the `datetime()` function.

Reading temporal types

When reading temporal types from the database, you will receive an instance of the corresponding Java type.

java:

```java
var result = driver.executableQuery("""
    RETURN date() as date, time() as time,
        datetime() as datetime,
        toString(datetime()) as asString;
    """)
    .execute();

var records = result.records();
records.forEach(r -> {
    System.out.println(r.get("date"));      // neo4j.time.Date
    System.out.println(r.get("time"));      // neo4j.time.Time
    System.out.println(r.get("datetime"));  // neo4j.time.DateTime
    System.out.println(r.get("asString"));  // String
});
```

Working with Durations

java:

```
import java.time.LocalDateTime;
import java.time.Duration;
var startsAt = LocalDateTime.now();
var eventLength = Duration.ofHours(1).plusMinutes(30);
var endsAt = startsAt.plus(eventLength);
var result = driver.executableQuery("""
    CREATE (e:Event { startsAt: $startsAt, endsAt: $endsAt,
        duration: $eventLength, // (1)
        interval: duration("PT1H30M") // (2)
    })
    RETURN e
    """)
    .withParameters(Map.of(
        "startsAt", startsAt, "endsAt", endsAt, "eventLength", eventLength
    ))
    .execute();
```

Durations represent a period of time and can be used for date arithmetic in both Java and Cypher. These types can also be created in Java or cast within a Cypher statement.

1. Pass an instance of Java `Duration` to the query
2. Use the `duration()` Cypher function to create a `Duration` object from an ISO 8601 format string

Calculating durations

You can use the `duration.between` method to calculate the duration between two date or time objects.

Spatial types

Points and locations

Neo4j has built-in support for two-dimensional and three-dimensional spatial data types. These are referred to as **points**.

A point may represent geographic coordinates (longitude, latitude) or Cartesian coordinates (x, y).

In Java, points are represented by the `org.neo4j.driver.types.Point` type, which is wrapped by the `org.neo4j.driver.Values` class to expose as a generic `Value` object.

The `Point` type provides methods to access the coordinates and SRID of the point, allowing for easy manipulation and retrieval of spatial data.

Cypher Type	Java Type	SRID	3D SRID
Point (Cartesian)	`org.neo4j.driver.types.Point`	`7203`	`9157`
Point (WGS-84)	`org.neo4j.driver.types.Point`	`4326`	`4979`

CartesianPoint

A Cartesian Point defines a point with x and y coordinates. An additional z value can be provided to define a three-dimensional point.

You can create a cartesian point by passing `x`, `y` and optionally `z` values to the `Values.point()` method:

java: **CartesianPoint**

```
import org.neo4j.driver.Values;


var location2d = Values.point(srid, x, y);
var location3d = Values.point(srid, x, y, z);
```

CartesianPoint

Points returned from Cypher queries are converted to instances of the `Point` interface:

java:

```
var result = driver.executableQuery("RETURN point({x: 1.23, y: 4.56, z: 7.89}) AS point")
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
var point = result.records().get(0).get("point");


System.out.println(point);
System.out.println(
    point.asPoint().x() + ", " + point.asPoint().y() + ", " + point.asPoint().z()
    );
```

The values can be accessed using the `x`, `y` and `z` methods.

WGS84Point

A WSG *(World Geodetic System)* point consists of a `latitude` (`y`) and `longitude` (`x`) value. An additional `height` (`z`) value can be provided to define a three-dimensional point.

You can create a WGS84 point by passing `longitude`, `latitude` and `height` values to the point function in Cypher or passing the values to the `Values.point()` in Java.

java: WGS84Point

```
import org.neo4j.driver.Values;


var location2d = Values.point(4326, -0.118092, 51.509865);
System.out.println(location2d.asPoint().x() + ", " +
                   location2d.asPoint().y() + ", " +
                   location2d.asPoint().srid());


var location3d = Values.point(4979, -0.086500, 51.504501, 310);
System.out.println(location3d.asPoint().x() + ", " +
                   location3d.asPoint().y() + ", " +
                   location3d.asPoint().z() + ", " +
                   location3d.asPoint().srid());
```

WGS84Point

The driver will return `WGS84Point` objects when `point` data types are created with `latitude` and `longitude` values in Cypher. The values can be accessed using the `x`, `y` and `z` attributes.

java: Using point()

```
var result = driver.executableQuery("""
        RETURN point(
            {latitude: 51.5, longitude: -0.118, height: 100}
        ) AS point
    """)
    .execute();

var point = result.records().get(0).get("point");
var longitude = point.asPoint().x();
var latitude = point.asPoint().y();
var height = point.asPoint().z();
var srid = point.asPoint().srid();

System.out.println(longitude + ", " + latitude + ", " + height + ", " + sri
System.out.println(point.asPoint());
```

Distance

The `point.distance` function can be used to calculate the distance between two points with the same SRID.

The result is a `float` representing the distance in a straight line between the two points.

SRIDs must be compatible

If the SRID values are different, the function will return `None`.

Java:

```
var point1 = Values.point(7203, 1.23, 4.56);
var point2 = Values.point(7203, 2.34, 5.67);

var result = driver.executableQuery("""
        RETURN point.distance($p1, $p2) AS distance
        """)
        .withParameters(
            Map.of("p1", point1, "p2", point2))
        .execute();

var distance = result.records().get(0).get("distance").asDouble();
System.out.println(distance);
```

== Check your understanding

Transaction management

Introduction

You have learned how to execute one-off Cypher statements using the `executableQuery()` method.

The drawback of this method is that the entire record set is only available once the final result is returned. For longer running queries or larger datasets, this can consume a lot of memory and a long wait for the final result.

In a production application, you may also need finer control of database transactions or to run multiple related queries as part of a single transaction.

Transaction methods allow you to run multiple queries in a single transaction while accessing results immediately.

Understanding Transactions

Neo4j is an ACID-compliant transactional database, which means queries are executed as part of a single atomic transaction. This ensures your data operations are consistent and reliable.

Sessions

To execute transactions, you need to open a session. The session object manages the underlying database connections and provides methods for executing transactions. For async applications, **use the** `AsyncSession` .

Java:

```
try (var session = driver.session()) {
    // Call transaction functions here
}
```

Consuming a session within a `try-with-resources` will automatically close the session and release any underlying connections when the block is exited.

Transaction functions

The `Session` object provides two methods for managing transactions:

- `Session.executeRead()`
- `Session.executeWrite()`

If the entire function runs successfully, the transaction is committed automatically. If any errors occur, the entire transaction is rolled back.

> **Transient errors**
>
> These functions will also retry if the transaction fails due to a transient error, for example, a network issue.

Unit of work patterns

A unit of work groups operations into a single method, which is executed using the `Session`:

Java:

```
// Unit of work
public static int createPerson(TransactionContext tx, String name, int age) { // (1)
    var result = tx.run("""
        CREATE (p:Person {name: $name, age: $age}) RETURN p
        """, Map.of("name", name, "age", age)); // (2)
    return result.list().size();
}
// Execute the unit of work
try (var session = driver.session()) { // (3)
    var count = session.executeWrite(tx -> createPerson(tx, name, age));
}
```

1. The first argument to the transaction function is always a `TransactionContext` object. Any additional arguments are passed from the call to `Session.executeRead` / `Session.executeWrite`.
2. The `run()` method on the `TransactionContext` object is called to execute a Cypher statement.
3. The `executeWrite()` method is called on the session object to execute the transaction function. The result of the transaction function is returned to the caller.

Multiple Queries in One Transaction

You can execute multiple queries within the same transaction function to ensure that all operations are completed or fail as a single unit.

Java:

```
public static void transferFunds(TransactionContext tx, String fromAccount, String toAccount, double amount) {
    tx.run(
        "MATCH (a:Account {id: $from_}) SET a.balance = a.balance - $amount",
        Map.of("from_", fromAccount, "amount", amount)
    );
    tx.run(
        "MATCH (a:Account {id: $to_}) SET a.balance = a.balance + $amount",
        Map.of("to_", toAccount, "amount", amount)
    );
}
```

Handling outputs

The `TransactionContext.run()` method returns a `Result` object.

The records contained within the result will be iterated over as soon as they are available.

The result must be consumed within the transaction function.

The `consume()` method discards any remaining records and returns a `ResultSummary` object that can be used to access metadata about the Cypher statement.

The `Session.executeRead` / `Session.executeWrite` method will return the result of the transaction function upon successful execution.

Java: Consuming results

```
public static ResultSummary getAnswer(TransactionContext tx, String answer) {
        var result = tx.run("RETURN $answer AS answer", Map.of("answer", answer));
        return result.consume();
    }


String result = "Hello, World!";
try (var session = driver.session()) {
    ResultSummary summary = session.executeWrite(tx -> getAnswer(tx, result));
    System.out.println(
        String.format(
            "Results available after %d ms and consumed after %d ms",
            summary.resultAvailableAfter(TimeUnit.MILLISECONDS),
            summary.resultConsumedAfter(TimeUnit.MILLISECONDS)
        )
    );
}
```

Handling Database Errors

Error Handling

When working with Neo4j, you may encounter various database errors that need to be handled gracefully in your application. The driver exports a `Neo4jException` **class** that is inherited by all exceptions thrown by the database.

Common exceptions

- `AuthenticationException` - Raised when authentication fails (incorrect credentials provided)
- `ClientException` - Raised when the client-side error occurs
- `ConnectionReadTimeoutException` - Raised when the connection to the database times out (transaction took longer than timeout - 30 seconds, by default)
- `DatabaseException` - Raised when there is a problem with the database
- `GqlStatusErrorClassification` - Raises an error based on GQL status codes
- `RetryableException` - Indicates retrying the transaction may succeed
- `ServiceUnavailableException` - Raised when the database is unavailable (e.g., server is down or not running)
- `TransactionTerminatedException` - Indicates that a transaction was terminated for some reason

Handling errors

When using the Neo4j Java driver, you can handle errors by catching specific exceptions. Any errors raised by the DBMS (`Neo4jException`) will have a `message` property that describes the error, as well as optional properties for `code` , `gql_status` , `cause` , and more.

java:

```java
try (var session = driver.session()) {
    // Run a Cypher statement
    var result = session.run("MATCH (n) RETURN n LIMIT 10;");
    result.forEachRemaining(record -> {
        System.out.println(record.get("n").asNode().asMap());
    });
} catch (Neo4jException e) {
    e.code(); // Outputs the error code
    e.getMessage(); // Outputs the error message
    e.gqlStatus(); // Outputs the GQL status
    e.printStackTrace(); // Outputs full stack trace
}
```

The `gqlStatus` property contains an error code that corresponds to an error in the ISO GQL standard. A full list of GQL Status codes can be found in [Status Codes for Errors & Notifications](#).

Example: Handling unique constraint violations

One common scenario is dealing with constraint violations when inserting data. A unique constraint ensures that a property value is unique across all nodes with a specific label.

The following Cypher statement creates a unique constraint named `unique_email` to ensure that the `email` property is unique for the `User` label:

cypher:

```
CREATE CONSTRAINT unique_email IF NOT EXISTS
FOR (u:User) REQUIRE u.email IS UNIQUE;
```

If a Cypher statement violates this constraint, Neo4j will raise a `ConstraintError`.

Java:

```
var name = "Test Name";
var email = "test@test.com";


try (var session = driver.session()) {
    var result = session.run("""
        CREATE (u:User {name: $name, email: $email})
        RETURN u;
    """,
    Values.parameters("name", name, "email", email));
} catch (ClientException e) {
    e.printStackTrace();
    // org.neo4j.driver.exceptions.ClientException:
        // Node(5) already exists with label `User`
        // and property `email` = 'test@test.com'
}
```